Integrating Cache Performance Modeling and Tuning Support in Parallelization Tools

Abdul Waheed and Jerry Yan
MRJ Technology Solutions
NASA Ames Research Center
Moffett Field, CA 94035-1000
E-mail: {waheed,yan}@nas.nasa.gov

Abstract

With the resurgence of distributed shared memory (DSM) systems based on cache-coherent Non Uniform Memory Access (ccNUMA) architectures and increasing disparity between memory and processors speeds, data locality overheads are becoming the greatest bottlenecks in the way of realizing potential high performance of these systems. While parallelization tools and compilers facilitate the users in porting their sequential applications to a DSM system, a lot of time and effort is needed to tune the memory performance of these applications to achieve reasonable speedup. In this paper, we show that integrating cache performance modeling and tuning support within a parallelization environment can alleviate this problem. The Cache Performance Modeling and Prediction Tool (CPMP), employs trace-driven simulation techniques without the overhead of generating and managing detailed address traces. CPMP predicts the cache performance impact of source code level "what-if" modifications in a program to assist a user in the tuning process. CPMP is built on top of a customized version of the Computer Aided Parallelization Tools (CAPTools) environment. Finally, we demonstrate how CPMP can be applied to tune a real Computational Fluid Dynamics (CFD) application.

1 Introduction

Distributed shared memory (DSM) multiprocessors offer ease of programming due to a global address space. A majority of commercial DSM multiprocessors employs Non Uniform Memory Access (NUMA) architecture for scalability purposes [19]. In addition, these multiprocessors are built around commodity parts, including processors, with one or more levels of caches. Therefore, a programmer has to deal with multiple levels of memory hierarchy to avoid memory performance bottlenecks in an application program. Due to increasing disparity between processor and memory performance, it is essential to enhance the utilization of caches to realize high performance potential of these multiprocessors. While global address space facilitates the task of a programmer to port a sequential application, a lot of effort is still needed to tune the cache performance.

In general, existing cache performance tuning approaches fall in one of two categories: measurement based and modeling based. Table 1 lists the analysis goals and limitations of measurement and modeling based cache performance evaluation techniques. None of these methodologies can be directly applied to assisting an application developer to tune memory performance of a source code. Level of effort and turn-around

time prohibit a user to apply modeling based approaches for tuning any real code. Nevertheless, trace-driven simulation approaches are considered reliable and accurate under realistic conditions [14].

Table 1. Various memory subsystem performance analysis techniques, their goals, and limitations with respect to application cache performance tuning.

Methodology	Analysis goals	Limitations
Measurements using on-chip performance counters	*Application profiling and cache measurements for identifying bottlenecks [3,28]*	*Excessive overhead for tracing due to kernel level interface; non-repeatable; and lack of "what-if" analysis support [27]*
Analytic modeling	*Performance projections for existing applications on future architectures [5,9,15,21]*	*Level of effort is inappropriate for analyzing "what-if" modifications for tuning*
Trace-driven simulation	*Accurate analysis of reference behavior for design and analysis of architectural features [4,10]*	*Generating and managing traces for even a moderate-sized block of a real application is non-trivial*
Execution-driven simulation	*Detailed system level performance evaluation of actual workload at various design stages [18]*	*Turn-around time is too long to be applicable in a tuning scenario*
Complete machine simulation	*Simulation of interaction between architecture and operating system level behavior [26]*	*Level of detail and effort involved in setting up the simulation environment make it inappropriate for tuning*

In this paper, we present an implementation of a uniprocessor cache performance tuning methodology that combines measurement and trace-driven simulation techniques. The Cache Performance Modeling and Prediction Tool, or "CPMP", retains the advantages of both techniques while avoiding their limitations. We have built CPMP on top of a parallelization tool, called *Computer Aided Parallelization Tools* (CAPTools [16]). Initial measurements help locate memory-intensive segments of the code. CPMP first constructs the memory model related to a selected code block. This model can be used to study the impact of various modifications in that code block on cache performance. User can select a modification that results in best cache performance for implementation in tuned version of the code. CPMP analyzes an annotated (a CAPTools generated) parse-tree representation of a Fortran77 source code and produces a corresponding simulation model. Initial minimal measurements generate information about base virtual addresses of arrays and variables and loop bounds in selected code block that may not be known statically. CPMP uses this information to accurately predict the memory reference behavior. Using this model, a user can predict cache performance with respect to coding alternatives and select the most suitable ones for an application. Scope of our discussion in this paper is restricted to on-chip (level one or primary) cache performance.

Section 2 motivates the need for uniprocessor cache optimization to obtain scalable multiprocessor performance and overviews our integration of CPMP in CAPTools environment. We focus on the implementation of CPMP in Section 3. In Section 4, we present a case study where CPMP is used for tuning cache performance of a Computational Fluid Dynamics (CFD) application, ARC3D, on an SGI

Origin2000. We review the research efforts related to our work in Section 5. We conclude with a discussion of integrating cache performance modeling and tuning with parallelizing tools and future directions of our work.

2 Integrated Parallelization and Modeling Environment

In this section, we first motivate the need for uniprocessor cache performance tuning using three programs from NAS Parallel Benchmarks suite. Subsequently, we describe the implementation of CPMP as an automatic modeling tool, which can be used in conjunction with parallelizing programs for a DSM system.

2.1 Parallelization for DSM Multiprocessors

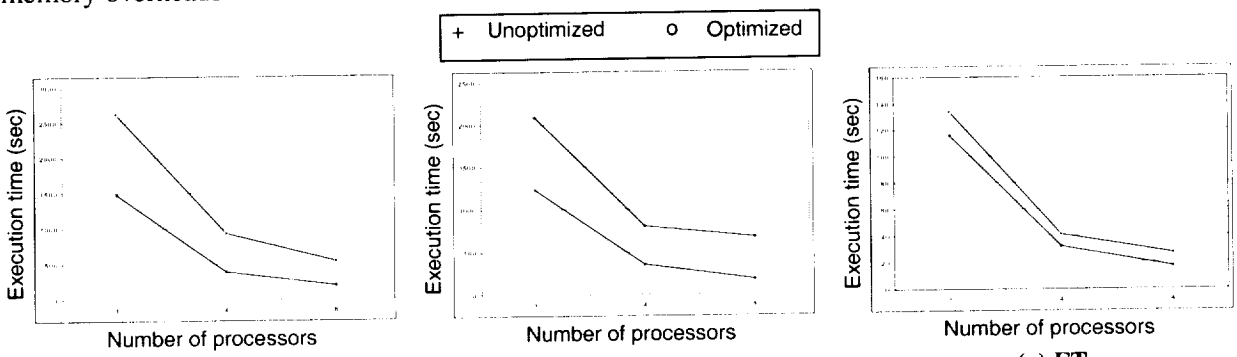
There are several paradigms that can be followed to parallelize sequential code for a DSM multiprocessor. These paradigms include: explicit message passing; data parallel programming; and shared memory programming. Shared memory programming is the simplest of these paradigms to implement in a parallelizing tool or compiler. Parallelization is based on loops that do not have any loop-carried data dependences among iterations. Loop iterations can be scheduled on multiple processors in a fork-and-join manner. CAPTools can analyze the source code to identify parallelizable loops. Parallelization directives are inserted to indicate to the compiler that the loops should be executed in parallel. During compilation, the compiler replaces the directives with appropriate runtime system calls for shared memory multiprocessing of loop iterations. Figure 1(a) shows a code segment taken from the backsubstitution phase of sequential implementation of the application benchmark BT. Using a customized version of CAPTools, we parallelize BT using OpenMP directives for shared memory multiprocessing. Using standards, such as OpenMP [24], ensures portability of the parallelized code to multiple shared memory systems. Figure 1(b) shows the CAPTools parallelized version of the code segment shown in Figure 1(a).

```
      do k=grid_points(3)-2,0,-1
      do j=1,grid_points(2)-2
      do i=1,grid_points(1)-2
      do m=1,BLOCK_SIZE
      do n=1,BLOCK_SIZE
        rhs(m,i,j,k) = rhs(m,i,j,k)
     >          - lhs(m,n,cc,i,j,k)*rhs(n,i,j,k+1)
       enddo
      enddo
      enddo                                    (a)
      enddo
      enddo
```

```
      c$omp paralleldo private(k,j,i,m,n)
      do k=grid_points(3)-2,0,-1
      do j=1,grid_points(2)-2
      do i=1,grid_points(1)-2
      do m=1,BLOCK_SIZE
      do n=1,BLOCK_SIZE
        rhs(m,i,j,k) = rhs(m,i,j,k)
     >          - lhs(m,n,cc,i,j,k)*rhs(n,i,j,k+1)
       enddo
      enddo
      enddo                                    (b)
      enddo
      enddo
```

Figure 1. Shared memory multiprocessing directives based parallelization of z_backsubstitute subroutine of BT. (a) Sequential code. (b) CAPTools parallelized code using OpenMP directives.

The above example underscores the minimal effort required on the part of user to parallelize the code for a shared memory system. Apparently, this parallelization process does not consider the memory hierarchy of the target system and hence cannot guarantee even reasonable speedup due to potential memory performance bottlenecks. We parallelized BT as well as several other benchmarks from NAS suite by inserting shared memory multiprocessing directives using a customized implementation of CAPTools and executed them on an Origin2000 system. Figure 2 compares the performance of optimized and unoptimized versions of three benchmarks: BT, SP, and FT. Despite parallelizing most of the loops in original versions of BT and SP benchmarks, the multiprocessor performance does not scale well due to memory overheads.



(a) BT **(b) SP** **(c) FT**

Figure 2. Impact of uniprocessor cache optimizations on multiprocessor performance for three benchmarks taken from Class A of NAS Parallel Benchmark suite.

We tuned the uniprocessor cache performance of sequential versions of BT and SP by minimizing the dimensions (i.e., sizes) of many temporary arrays. The original codes were written for vector supercomputers where larger temporary arrays are recommended to fully benefit from vector registers. However, larger temporary arrays result in excessive cache contention and misses on a cache-based processor. Minimizing array sizes requires extensive modifications in the code by a user who is also knowledgeable about the algorithm. Figure 3 presents the relevant parts of optimized version of the code shown in Figure 1. Parallelized version of this code also privatizes the `lhs` array that further improves data locality. Therefore, after uniprocessor cache performance tuning, scalability of parallelized BT and SP on multiple processors is close to linear (see Figure 2). In case of FT, optimization is related to loop nest transformations to enhance the parallel coverage of the program with additional loop level parallelism.

These examples indicate the importance of uniprocessor cache performance tuning for a shared memory parallel program. Unfortunately, cache performance tuning is an iterative process and may not be completely automated similar to the shared memory parallelization process. Therefore, it is important to have memory performance modeling tools that predict the impact of alternative source code modifications on cache performance.

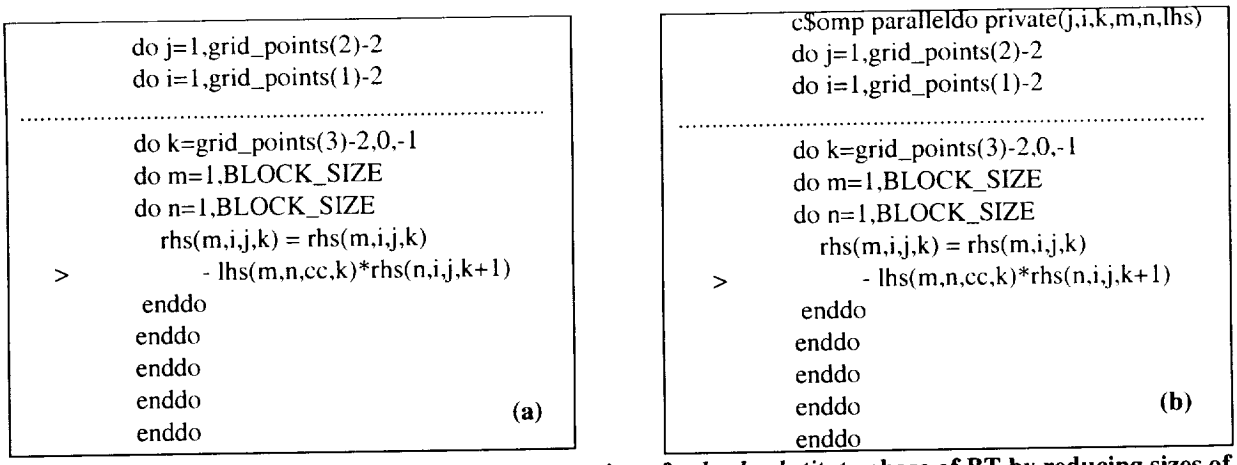


Figure 3. Uniprocessor cache performance tuning of *z_backsubstitute* phase of BT by reducing sizes of temporary arrays and privatizing them. (a) Sequential code. (b) CAPTools parallelized code.

2.2 Integration

In order to integrate cache performance modeling with parallelization process, we rely on an annotated parse-tree of the code created by CAPTools. Figure 4 provides an overview of integrating cache performance modeling support in CAPTools parallelization environment. Initial measurements are needed to obtain runtime information to parameterize a selected code block. An automatic model generator then uses the parse-tree of the source code and measured parameters to generate a simulation model of memory references. This model is linked with a runtime library of a cache, which is parameterized for a particular target system. Executing this model provides cache miss statistics. Comparing these cache miss statistics for alternative code modifications, a user can determine the most suitable modification to be incorporated in the original source code.

3 Automatic Cache Performance Model Generation

Automatic cache performance model generation for a Fortran77 source code block is based on the memory references found in the parse-tree representation of that code. In this section, we explain the model generation process starting from source to a memory model through its parse-tree representation in CAPTools using an example code block shown in Figure 5. We focus our attention to only basic blocks of code in which control flow does not change. While a DO statement is permissible, we assume that a selected block of code does not contain any subroutine calls or IF constructs. These assumptions are not overly restrictive as many numerical problems consist of memory-intensive kernels that are implemented as basic blocks. There are three constructs of a basic block that need further attention to details: assignment statements, array references, and DO loops.

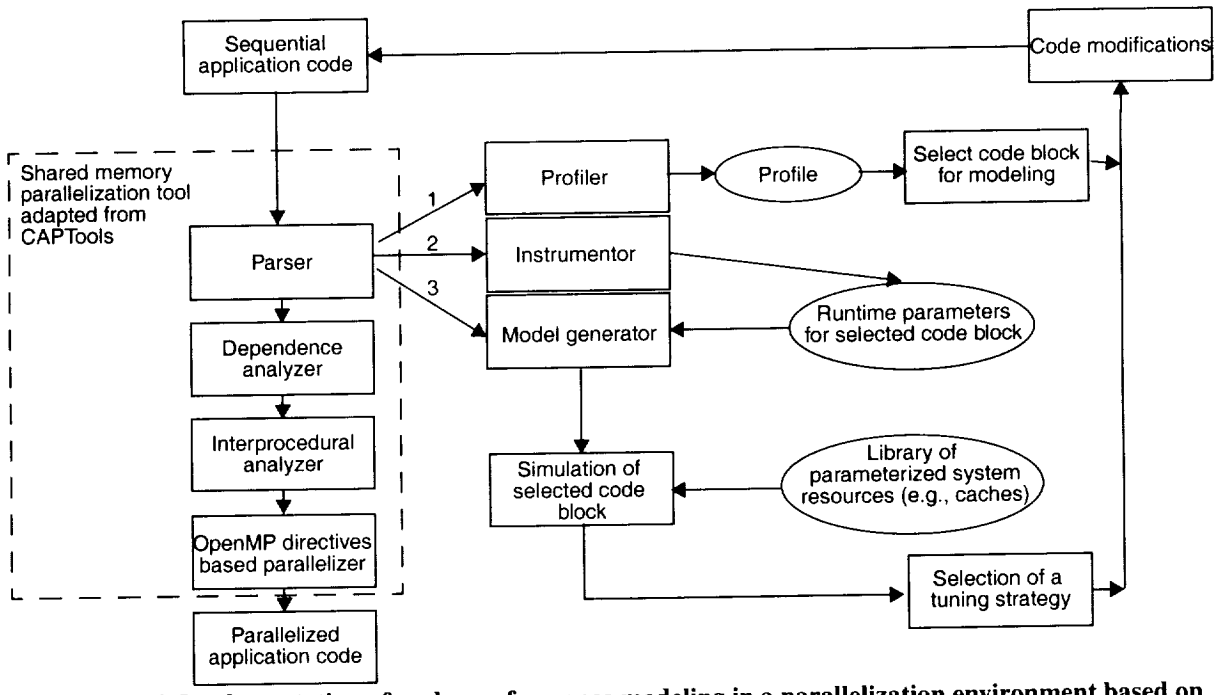


Figure 4. Implementation of cache performance modeling in a parallelization environment based on CAPTools for tuning uniprocessor cache performance.

```
parameter (nx=64, ny=64)
real a(0:64, 0:64)
real b(nx,ny), c(nx,ny)
real d

do i = 1,nx
     do j = ny,1,-1
          a(i,j+1) = b(i,j)*c(j,k) + d
     enddo
enddo
```

Figure 5. An example code block with three Fortran77 constructs of interest: assignment statement, array references, and DO loops.

3.1 Assignment Statements

In the absence of function calls, an assignment statement is the only obvious way to access memory to accomplish various computations. Memory accesses may also be needed for updating indices of a DO loop. However, our experience with modeling several CFD applications indicates that the number and impact of such references on overall cache performance is insignificant. Additionally, several compilers for RISC processors use register variables for array indices to eliminate the need for memory accesses for updating or reading array index values. Therefore, an assignment statement is a major source of memory references in a Fortran77 program.

Using a parse-tree representation of a basic block, CPMP first identifies assignment statements and then extracts array and variable references. Figure 6(a) presents a typical assignment statement involving

reading from two array elements and one variable and then writing the results to another array element. Figure 6(b) presents the parse-tree representation of that assignment statement. Analysis of this parse-tree can help identify: (1) read and write memory accesses based on the placement of an access on right or left sides of an equality, respectively; and (2) variable and array accesses as the node that represents an array name has descendents to identify array indices. Figure 6(c) depicts the part of cache performance model corresponding to the assignment statement shown in Figure 6(a). Size of each reference is extracted from their definition, a process which is explained in the following subsection.

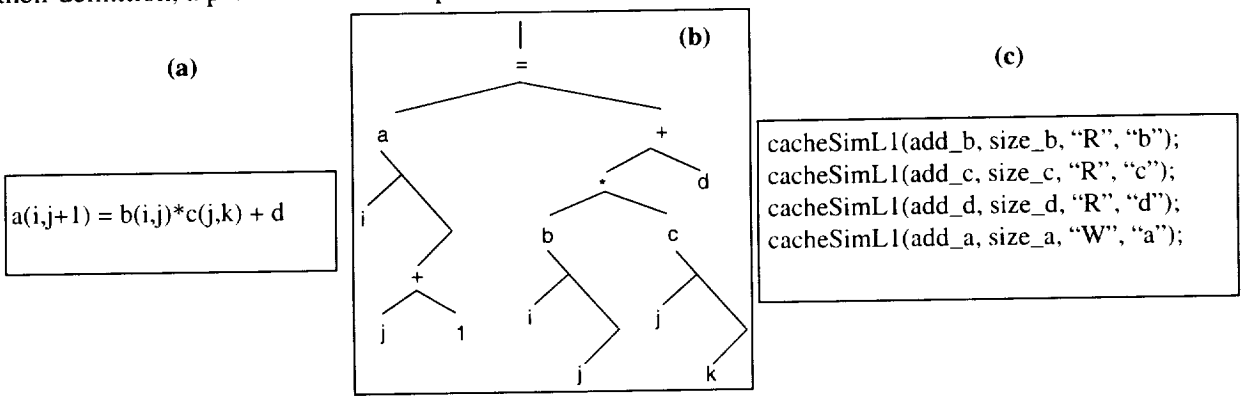


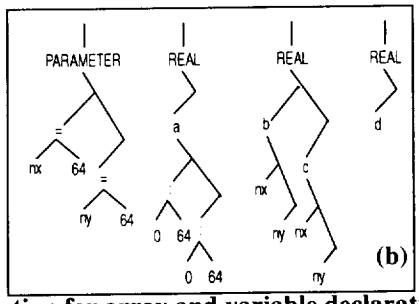
Figure 6. An example of automatic model generation. (a) An assignment statement in Fortran77; (b) parse-tree representation of the assignment statement; and (c) simplified generated model for the statement. Actual generated model contains a formula to calculate array reference addresses as a function of the base address and indices for that array references rather than a single variable.

3.2 Array References

Since Fortran programs are often used for scientific computation of numerical algorithms, a number of array access are expected in such programs. Arrays store program data in contiguous memory location of identical sizes, which are determined by array type declaration in a program. Figure 7 highlights the steps involved in generating necessary cache modeling code from array and variable declarations encountered in the example Fortran77 code. If an array or variable is encountered in a statement in a selected code block, we look for their declarations in the current subroutine, such as those presented in Figure 7(a). After finding those declarations in the annotated parse-tree as shown in Figure 7(b), CPMP generates the necessary code to define additional data structures to complement rest of the cache performance model for that code block. This generated code is shown in Figure 7(c).

Model generation for references to array elements is different from scalar references in terms of computation of the virtual address. While the virtual address of a scalar reference can be measured once, we may have to determine addresses of individual elements for an array. Using Fortran convention of storing an array in a column-major fashion, the address of an array element $A(I,J,K)$ is calculated with respect to the base address of $A(1,1,1)$ as:

```
parameter (nx=64, ny=64)
real a(0:64, 0:64)
real b(nx,ny), c(nx,ny)
real d
```
(a)



(b)

```
int dim_a[2] = {65,65};
int dim_b[2] = {64,64};
int dim_c[2] = {64,64};
int size_a = 8;
int size_b = 8;
int size_c = 8;
int size_d = 8;
```
(c)

Figure 7. Automatic code generation for array and variable declarations in a Fortran77 program. (a) A code segment showing some declarations. (b) Parse-tree representation of four statements. (c) Generated code for cache performance modeling corresponding to the four statements.

$$Address(A(I, J, K)) = Address(A(1, 1, 1)) + (I - 1) + J\text{dim\_A}(1) + K(\text{dim\_A}(1).\text{dim\_A}(2)), \quad (1)$$

where *dim_A* is a three dimensional vector such that each dimension specifies the size of corresponding dimension of array *A*.

3.3 DO Loops

One characteristic of a numerical algorithm is its repetition of a core set of statements to accomplish an iterative computation. This characteristic manifests itself in a Fortran77 program in the form of DO loops. Therefore, DO loops are considered an important construct in a scientific application for several software tools, including parallelizing compilers. In the context of memory model generation, DO loops are important because they represent a repetitive set of memory references. If some of these repetitive references are array elements, their address is calculated by generalizing the equation (1) for each iteration of the loop. Repeated accesses to a set of memory locations within a DO loop are modeled with repetitions of modeled accesses in each iteration of the loop.

Figure 8 illustrates the process of automatically generating cache performance model code from a loop nest in the example Fortran77 code. In this particular case, values of the symbols nx and ny, which are used as loop bounds, are determined from their declarations in a parameter statement using the process presented in Figure 7. In other cases, it may not be possible to fully determine loop bounds and step values statically. In those cases, we rely on the information gathered at runtime.

Using the model generation processes for three Fortran77 constructs, CPMP automatically generates cache performance model for any basic block found in a program. We used Dinero trace-driven simulation tool to validate the functionality and results of this model generator [8]. Details of this validation process will be presented in the full paper.

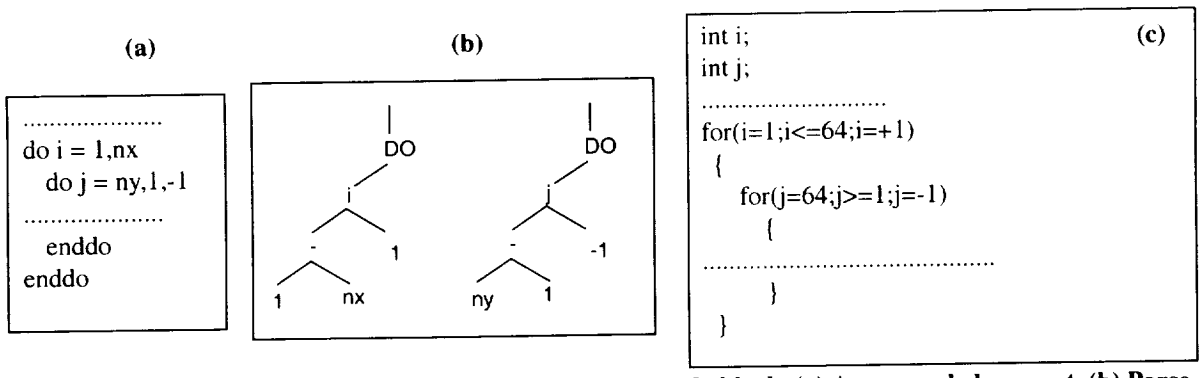


Figure 8. Code generation for DO loops in a Fortran77 code block. (a) An example loop nest. (b) Parse-tree representation for the example DO loops. (c) Generate code for cache performance modeling. Note that loop bounds are determined from the process illustrated in Figure 7.

4 Application Cache Performance Modeling and Tuning: A Case Study

In this section, we briefly present our experience of applying CPMP for modeling and tuning ARC3D. ARC3D is a CFD application that solves a system of Naviar-Stokes partial differential equations for a three dimensional mesh using scalar pentagonal algorithm. The original sequential code is written for vector supercomputers and our objective is to port and tune it for an Origin2000 system. Based on initial measurements of single processor execution of ARC3D, we decided to focus on the solver part of the application. Figure 9 presents a code segment adapted from sequential implementation of solver phase that works along x-direction (to be referred to as RHSX). Due to the complexity of this code, it is tedious to try to manually generate a cache performance model for this code. We apply the automatic cache performance model generator to this code block.

Initial measurements provides necessary base address and loop bound information which complements the rest of the information obtained from source code analysis. Tables 2 and 3 provide measurement and source code analysis based information obtained from the RHSX code block for its cache performance modeling. Note that the column indicating array indices in Table 2 represents the array index for the first reference involving that array. Some array references have different indices in subsequent references for this code block. Also note that the loop bounds information is completely specified by source code analysis in Table 3. However, in many other cases only runtime measurements may supply this information. Several modifications can be implemented in the original source code in an attempt to improve cache performance [25]. Some of these coding alternatives are listed in Table 4.

Figure 10 compares the cache performance due to six alternative modifications of the original code in terms of number of *cache misses* and *cache miss ratio*, which is a ratio of the number of misses to total number of memory references. Measurement based cache statistics are obtained through the *perfex* tool on

```
 real xxx(64,64,64), xxy(64,64,64), xxz(64,64,64)
 real e(64,64,8), s(64,64,64,5),q(64,64,64,6)
 real qsx,pp,qsinfx,pinfj,uinf,vinf,winf,rx4
 integer j,k,l,n

 do k=2,64
  do l=1,64
   do j=1,64
    qsx = rx4 +
>        (xxx(j,k,l)*q(j,k,l,2) + xxy(j,k,l)*q(j,k,l,3) +
>        xxz(j,k,l)*q(j,k,l,4))/q(j,k,l,1)
    pp = (q(j,k,l,2)*q(j,k,l,2)+q(j,k,l,3)*q(j,k,l,3)+
>        q(j,k,l,4)*q(j,k,l,4))*0.5/q(j,k,l,1)
    qsinfx = (rx4+xxx(j,k,l)*uinf+xxy(j,k,l)*vinf+
>        xxz(j,k,l)*winf)*(1.0/q(j,k,l,6))
    pinfj = (1.0/q(j,k,l,6))*1.4
    e(j,l,1) = q(j,k,l,1)*qsx - qsinfx
    e(j,l,2) = q(j,k,l,2)*qsx + xxx(j,k,l)*pp -
>        uinf*qsinfx - xxx(j,k,l)*pinfj
    e(j,l,3) = q(j,k,l,3)*qsx + xxy(j,k,l)*pp -
>        vinf*qsinfx - xxy(j,k,l)*pinfj
    e(j,l,4) = q(j,k,l,4)*qsx + xxz(j,k,l)*pp -
>        winf*qsinfx - xxz(j,k,l)*pinfj
    e(j,l,5) = (q(j,k,l,5)+pp)*qsx - qsinfx
   enddo
  enddo
```

```
  do n=1,5
   do j=2,64
    s(j,k,2,n) = (e(j,3,n)-e(j,1,n))*(-0.5)
     s(j,k,64,n)= (e(j,64,n)-e(j,63,n))
   enddo
  enddo

  do n=1,5
   do l=3,62
    do j=2,63
     s(j,k,l,n) = e(j,l+2,n)+e(j,l+1,n)+
>        e(j,l-1,n)+e(j,l-2,n)
    enddo
   enddo
  enddo

 enddo
 end
```

Figure 9. A code segment adapted from RHS solver in x-direction in ARC3D application.

Table 2. Selected memory reference information related to the RHSX phase obtained from measurements and source code parsing.

Source code parsing based information					Measurement based information
Reference symbol	**Reference type**	**Reference size**	**Array dimension**	**Array index**	**Base virtual address**
k	*Write*	*4*	—	—	*0X7FFF2EBC*
l	*Write*	*4*	—	—	*0X7FFF2EC0*
j	*Write*	*4*	—	—	*0X7FFF2EB4*
n	*Write*	*4*	—	—	*0X7FFF2EB0*
xxx	*Read*	*8*	*(64,64,64)*	*(j,k,l)*	*0X7FEF2EA0*
xxy	*Read*	*8*	*(64,64,64)*	*(j,k,l)*	*0X7F7F2EA0*
xxz	*Read*	*8*	*(64,64,64)*	*(j,k,l)*	*0X7F6F2EA0*
e	*Read/Write*	*8*	*(64,64,8)*	*(j,l,1)*	*0X7F6D2EA0*
s	*Write*	*8*	*(64,64,64,5)*	*(j,k,2,n)*	*0X7F1D2EA0*
q	*Read*	*8*	*(64,64,64,6)*	*(j,k,l,2)*	*0X7F8F2EA0*
qsx	*Read/Write*	*8*	—	—	*0X7FFF2EB0*

Origin2000, which are not accurate due to sampling and software multiplexing of two physical counters. Cache performance predictions indicate that following code modifications improve cache performance compared to the original code: array padding (#2), loop nest transformations (#4), reduction of temporary array sizes (#5), and blocking (#6). Finally, we combine these modifications that individually work in the

Table 3. Loop nest information related to the RHSX phase obtained from measurements and source code parsing.

Source code parsing based information					Measurement based information		
Loop level	Index variable	Lower bound	Upper bound	Step	Lower bound	Upper bound	Step
1	k	2	64	1	2	64	1
2	l	1	64	1	1	64	1
3	j	1	64	1	1	64	1
2	n	1	5	1	1	5	1
3	j	2	64	1	2	64	1
2	n	1	5	1	1	5	1
3	l	3	62	1	3	62	1
4	j	2	63	1	2	63	1

Table 4. Possible modifications of RHSX code.

Modification	Explanation
1	*Original code*
2	*Array padding to make dimensions of all arrays non-power-of-two values*
3	*Array restructuring for e, s, and q, such that their dimensions become e(8,64,64), s(5,64,64,64) and q(6,64,64,64)*
4	*Loop nest transformations to reduce stride with restructured arrays for modification # 3*
5	*Reduction of temporary array sizes by making major changes in code for "de-vectorizing"*
6	*Blocking by saving references for multiply accessed array elements in temporary variables*
7	*A combination of above techniques that individually result in cache performance improvement*

final tuned version of the code. Predicted cache performance corresponding to this version (#7) shows best improvement compared to the original code.

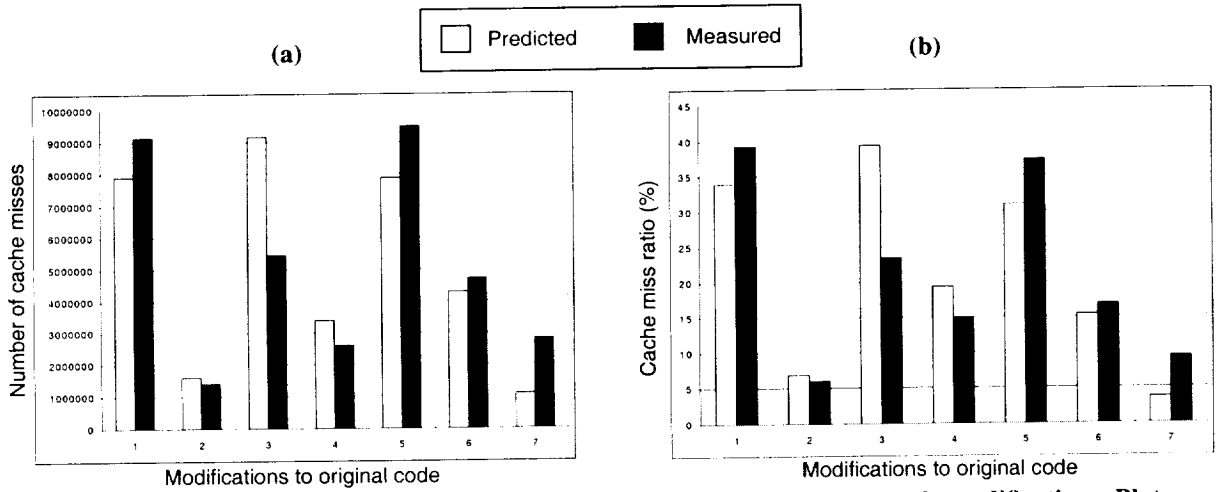


Figure 10. Predicted and measured cache performance due to various code modifications. Plots represent a comparison of (a) number of cache misses and (b) cache miss ratios.

We modified RHSY and RHSZ following the modifications implemented in RHSX. We parallelized this uniprocessor cache performance tuned version of ARC3D using our customized implementation of

CAPTools. Figure 11 presents the measured scalability characteristics of ARC3D on an Origin2000. Uniprocessor cache performance tuning results in about 80% reduction in execution time compared to original version. In addition, multiprocessor performance shows almost linear speedup.

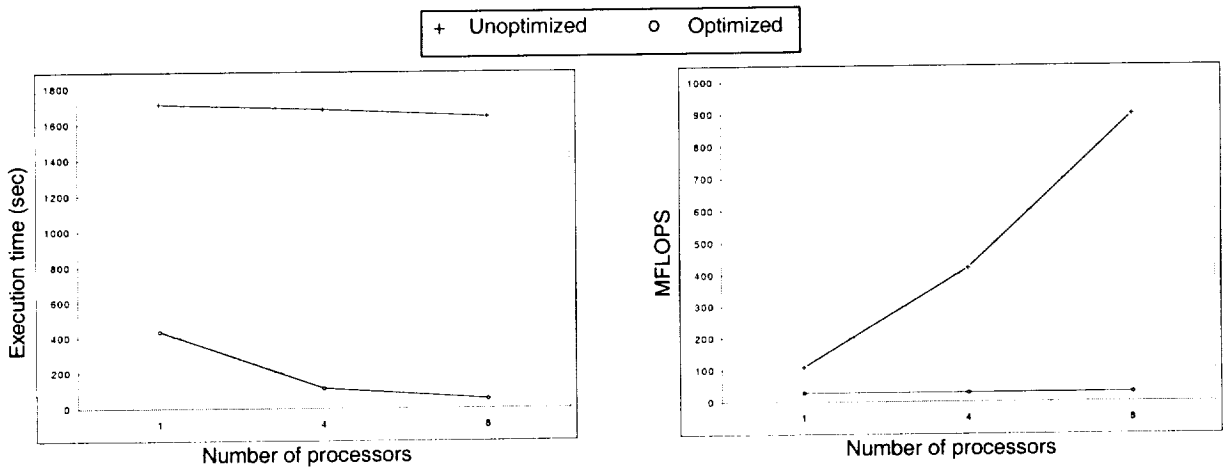


Figure 11. Comparison of multiprocessor performance of original and cache tuned versions of ARC3D on an Origin2000.

5 Related Work

Several cache performance modeling efforts have tried to combine multiple techniques for specific evaluation goals. Mtool combines low overhead instrumentation to generate enough information to isolate memory performance bottlenecks by analyzing the difference between actual and predicted cache performance [13]. Martonosi et al. have investigated the use of memory coherence protocol data in multiprocessors for analyzing memory performance [23]. MemSpy uses a trace-driven simulation with profiling to explore the causes of cache misses using traces generated by executing the instrumented programs [22]. Our experience indicates that generating detailed memory reference traces for an interesting code block from even a moderately complex application not only causes excessive perturbation but also generates unmanageable amounts of trace data.

A number of researchers are integrating compiler level information about source code with system models at different levels of detail for different analysis purposes [6]. Adve et al. explore the possibility of using compiler information statically as well as dynamically at runtime for architecture oriented tuning [1]. Compilers are integrated with measurement based performance evaluation tools for data parallel programs [2]. Ghosh et al. derive cache miss equations based on source code level analysis and use them in SUIF compiler system for cache performance tuning [11]. Mowry uses prefetching techniques to exploit latency hiding mechanisms to optimize memory reference locality [20]. These efforts indicate a growing trend of embedding performance analysis into the compiler to facilitate the tuning task for the end user. Our

implementation of CPMP in a parallelization environment is an initial practical step toward this goal for tuning application cache performance on multiprocessors.

6 Conclusions

In this paper, we presented CPMP, a cache performance modeling and prediction tool integrated in a parallelization environment. We demonstrated this tool based mostly on source code analysis and minimally on runtime information. The integrated environment was applied for parallelization and cache performance modeling and tuning of ARC3D. Measurements based results of optimized version of ARC3D showed the benefits of uniprocessor cache performance tuning for scalable multiprocessor performance.

Integration of CPMP in a parallelization tool is a step toward implementing cache performance modeling and tuning within a parallelization compiler. Complexity of memory subsystems in state-of-the-art parallel systems is making it increasingly difficult for a user to tune an application using existing measure-modify-execute approach. Many researchers believe that using strategically gathered runtime information, a compiler can play an active role in tuning an application [1]. CPMP relies on minimal amount of runtime information to maintain the accuracy of cache performance predictions. We are extending CPMP to use object code to determine base addresses relative to a dynamically allocated object, such as stack or heap. These estimated base addresses can become part of parameterization of a system. This approximation may affect accuracy of predictions but it will facilitate its seamless integration in a parallelization environment.

Acknowledgments

We thank our colleagues, H. Jin and J. Taft, for suggesting and actually implementing the cache optimizations in sequential implementations of NAS Parallel Benchmarks and ARC3D.

Bibliography

[1] Sarita V. Adve, Doug Burger, Rudolf Eigenmann, Alasdair Rawsthorne, Michael D. Smith, Catherine H. Gebotys, Mahmut T. Kandemir, David L. Lilija, Alok N. Choudhary, Jesse Z. Fang, and Pen-Chung Yew, "Changing Interaction of Compiler and Architecture," *IEEE Computer*, Dec. 1997, pp. 51–58.

[2] Vikram Adve, Jhy-Chun Wang, John Mellor-Crummey, Daniel Reed, Mark Anderson, and K. Kennedy, "An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs," *in the Proc. of Supercomputing* '95, San Diego, California, Dec. 1995.

[3] Glenn Ammons, Thomas Ball, and James R. Larus, "Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling," *in Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Las Vegas, Nevada, June 1997.

[4] Pradip Bose and Thomas M. Conte, "Performance Analysis and its Impact on Design," *IEEE Computer*, May 1998, pp. 41–49.

[5] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramanian, and Thorsten Von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," *in Proc. of the 4th Symposium on Principles and Practices of Parallel Programming (PPoPP '93)*, May 1993, pp. 1–12.

[6] Ewa Deelman, Aditya Dube, Adolfy Hoisie, Yong Luo, Richard Oliver, David Sunderam-Stukel, Harvey Wasserman, Vikram S. Adve, Rajive Bagrodia, James C. Browne, Elias Houstis, Olaf Lubeck, John Rice, Patricia Teller, and Mary K. Vernon, "POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems," *to Appear in the Proc. of the First International Workshop on Software and Performance*, Santa Fe, New Mexico, Oct. 1998.

[7] Eric van der Deijl, Gerco Kanbier, Olivier Temam, and Elena D. Granston, "A Cache Visualization Tool," *IEEE Computer*, 30(7), July 1997, pp. 71–78.

[8] Jan Edler and Mark D. Hill, "Dinero IV Trace-Driven Uniprocessor Cache Simulator," Available online from http://www.cs.wisc.edu/~markhill/DineroIV.

[9] Matthew I. Frank, Ananat Agarwal, and May K. Vernon, "LoPC: Modeling Contention in Parallel Algorithms," *in Proc. of the 6th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP '97)*, Las Vegas, Nevada, June. 1997, pp. 276–287.

[10] Jeffrey D. Gee, Mark D. Hill, Dionisios N. Pnevmatikatos, and Alan Jay Smith, "Cache Performance of the SPEC92 Benchmark Suite," *IEEE Micro*, August 1993.

[11] Somnath Ghosh, Margaret Martonosi, and Shard Malik, "Cache Miss Equations: An Analytical Representation of Cache Misses," *in Proc. of the 11th ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.

[12] Gideon Glass and Pei Cao, "Adaptive Page Replacement Based on Memory Reference Behavior," *in Proc. of Sigmetrics '97*, Seattle, Washington, June 1997.

[13] Aaron Goldberg and John Hennessy, "Mtool: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications," *IEEE Transactions on Parallel and Distributed Systems*, 4(1), Jan. 1993, pp. 28–40.

[14] Stephen R. Goldschmidt and John Hennessy, "The Accuracy of Trace-Driven Simulations of Multiprocessors," *in Proc. of Sigmetrics '95*, 1995, pp. 146–157.

[15] Adolfy Hoisie, Olaf Lubeck, and Harvey Wasserman, "Performance and Scalability Analysis of Teraflop-Scale Parallel Architectures Using Multidimensional Wavefront Applications," Technical Report, Los Alamos National Laboratory, Aug. 1998.

[16] C. S. Ierotheou, S. P. Johnson, M. Cross, and P. F. Leggett, "Computer Aided Parallelisation Tools (CAPTools)—Conceptual Overview and Performance on the Parallelisation of Structured Mesh Codes," *Parallel Computing*, Vo. 22, 1996, pp. 163–195.

[17] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith, "Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors," *in Proc. of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[18] J. Larus, "The SPIM Simulator for the MPIS R2000/R3000," in *Computer Organization and Design—The Hardware/Software Interface* by David A. Patterson and John L. Hennessy, Morgan Kaufmann Publishers, 1994.

[19] James Laudon and Daniel Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," *Proc. of the 24th Annual International Symposium on Computer Architecture*, Denver, Colorado, June 2-4, 1997, pp. 241–251.

[20] Todd C. Mowry, "Tolerating latency in multiprocessors through compiler-inserted prefetching," *ACM Transactions on Computer Systems*, 16(1), Feb. 1998, pp. 55–92.

[21] Yong Luo, Olaf M. Lubeck, Harvey Wasserman, Federico Bassetti, and Kirk W. Ameron, "Development and Validation of a Hierarchical Memory Model Incorporating CPU- and Memory-Operation Overlap," Technical Report, Los Alamos National Laboratory, Sept. 1998.

[22] Margaret Martonosi, Anoop Gupta, and Thomas E. Anderson, "Tuning Memory Performance of Sequential and Parallel Programs," *IEEE Computer*, 28(4), April 1995, pp. 32–40.

[23] Margaret Martonosi, David Oflet, and Mark Heinrich, "Integrating Performance Monitoring and Communication in Parallel Computers," *in Proc. of Sigmetrics '96*, Philadelphia, Pennsylvania, May 1996.

[24] *OpenMP Fortran Application Program Interface*, Available on-line from http://www.openmp.org, Oct. 97.

[25] *Optimization and Tuning Guide for Fortran, C, and C++ — AIX Version 3.2 for RISC System/6000*, IBM, 1993.

[26] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta, "Complete Computer Simulation: The SimOS Approach," *IEEE Parallel and Distributed Technology*, Fall 1995.

[27] Abdul Waheed and Jerry Yan, "Performance Measurement of Parallel and Distributed System Using On-Chip Counters," *accepted to appear in The International Journal of Parallel and Distributed Systems and Networks*, 1998.

[28] M. Zagha, B. Larson, Steve Turner, Marty Itzkowitz, "Performance Analysis Using the Mips R10000 Performance Counters," *in Proc. of Supercomputing '96*, Pittsburgh, Pennsylvania, Nov. 1996.